Custom Controls

Overview

Custom controls allow for a more customized appearance for a plugin. They also allow for the construction of more complex controls. One example of this would be a set of side-by-side dropdown menus to set column numbers for each breakpoint. At a basic level, Pinegrow provides two methods to create controls with a custom look - the `toggle_buttons` key plus `html` key with or without the `PgToggleButtonMaker()` constructor. At a more advanced level the Pinegrow API a series of keys and helper functions for novel layouts.

toggle_buttons

As briefly mentioned in the documentation of the `select` type, this key works along with the `html` key. This key takes a boolean value and if set to true it converts the dropdown list to side-by-side buttons. Reguires the use of the `html` key in the option list.

html

This key takes an HTML string of function that returns an HTML string as value and should be provided as the third key in an individual option object within an option array. This HTML string can contain inline styling and classes.

```
options: [
        {
                key: 'push',
                name: 'Push',
                html: '<div class="pg-tb-button pg-tb-button-text" style="font-weight: 700; color: white;" title="Push">Push</div>'
        },
        {
                key: 'slide',
                name: 'Slide',
                html: '<div class="pg-tb-button pg-tb-button-text" style="font-weight: 700; color: white;" title="Slide">Slide</div>
        }
        }
]
```

For simple buttons the `pg-tb-button` and `pg-tb-button-text` classes can be added to maintain an overall appearance that mimics the standard interface.

PgToggleButtonMaker()

This constructor exposes a number of different helper functions that aid in making different types of interface buttons. It doesn't receive any arguments and each of the helper functions are passed to the `html` key.

Basic usage

```
var button_maker = new PgToggleButtonMaker();
```

helper functions

makeText(text, options)

This helper function receives two arguments.

- **text** This argument accepts a string that will be displayed on the button.
- **options** This argument receives an object containing key:value pairs, where each value is an object composed of key:value pairs. The two primary keys are `styles` and `attributes`.
    - `styles`
      This key receives an object containing key:value pairs where each key is a valid CSS property, e.g. `color` or `padding`, and the value is what you want the property value set to, e.g. `red`. These styles will be added inline to the resulting button.
    - `attributes`
      This key receives an object containing key:value pairs where each key is an attribute name, e.g. `data-toggle` or `placement`, and the value is the attribute set value, e.g. `tooltip` or `left`.
      Example usage

```
'options' : [
    {
        'key' : 'normal',
        'name' : 'Normal',
        'html' : button_maker.makeText('Abc', {
            styles: {'font-family': 'serif', 'font-size': '15px'},
            attributes: {'title': 'Normal'}
        })
    },
    {
        'key' : 'small-caps',
        'name' : 'Small caps',
        'html' : button_maker.makeText('Abc', {
            styles: {'font-family': 'serif', 'font-size': '15px', 'font-variant': 'small-caps'},
            attributes: {'title': 'Small caps'}
        })
    }
]
```

Note: in this example a `title` attribute is being set. This will appear when the user hovers over the button.

Output of example:

makeColor(color, text, options)

This helper is of limited use due to the caveats noted below. It is usually better to use the makeText helper with additional styling for any button other than a solid color selector. This helper function returns a small 16px x 18px button with a colored background. It accepts three arguments.

- **color** This argument receives the desired background color of the button.
- **text** This argument receives an HTML string to be displayed on the button - with the caveat that the string must fit on a small button, usually a single character.
- **options** Like `makeText()`, this argument can receive a `styles` object and an `attribute` object.

The one caveat to using this helper is that any objects supplied through the `options` argument are merged with the existing key:value pairs set by default in the `makeColor()` function. This means that the default styling and passed in color argument will be overridden by any `style` object passed in. No default attributes are set in the function, so there is no conflict in passing an `attribute` object.

Typical usage:

```
{
        'key': 'primary',
        'name': 'Primary',
        'html' : button_maker.makeColor('#FF0000')
}
```

Example output from the Pinegrow Bootstrap Framework

makeColorText(color, options, text)

This helper function is of limited use due to the caveats noted below. It is usually better to use the makeText helper with additional styling for any button other than a solid color selector. It returns a small 16px x 18px button that is populated with an icon representing the letter "A" in the specified color. Formally, the function receives three arguments.

- **color** This argument receives the desired color of the icon displayed on the button.
- **options** This argument receives `styles` and `options` objects like the other helpers described above.
- **text** This argument can be passed to the function but is not used in the function.

Overall, this function has the same caveats as the `makeColor()` function.

Example output from Pinegrow Bootstrap Framework

makeIcon(icon, otions)

This helper function creates a small button displaying the passed icon. It receives two arguments.

- **icon** This argument is a string representing the name of the glyph to be displayed on the button.
- **options** This argument takes `settings` and `attributes` objects like the `makeText()` function.

Unlike the `makeText()` function, the `makeIcon()` function has no predefined options, so any that are passed will not impact the default styling. A table of the glyph names can be accessed here (https://bodonkey.github.io/Pinegrow-Fonts/). Please note that not all names are descriptive, some are simply numeric references. There is also a downloadable PDF file listing all of the glyphs and names that can be accessed here (https://raw.githubusercontent.com/BoDonkey/Pinegrow-Fonts/master/PG%20Fonts%20Cheatsheet.pdf).

Example usage:

```
options: [{
                key: 'left',
                name: 'Left',
                html: uikitButton.makeIcon('align-left', {
                        attributes: {
                                title: 'Left'
                        }
                })
        },
        {
                key: 'center',
                name: 'Center',
                html: uikitButton.makeIcon('align-center', {
                        attributes: {
                                title: 'Center'
                        }
                })
        },
        {
                key: 'right',
                name: 'Right',
                html: uikitButton.makeIcon('align-right', {
                        attributes: {
                                title: 'Right'
                        }
                })
        },
        {
                key: 'none',
                name: 'None',
                html: '<div class="pg-tb-button pg-tb-button-text" style="font-weight: 700; color: white;" title="None">X</div>'
        }
],
```

Advanced Custom Controls

Overview

Not only does the Pinegrow API provide standard controls for constructing your plugin UI, it also has several helper functions and constructors to allow the composition of custom controls with unique look. For example, the "Margin & Padding" control within the Pinegrow visual CSS editor is a custom control.

These custom controls are composed as functions that are called from the `control` key of one or more fields. The called function instantiates a new control using the `PgCustomPropertyControl` constructor. This constructor takes two keys, `registerInputField` and `showInputField`, that register and return return HTML for each of the control elements.

Example Usage

```
var pge_breakpoints = [ 'small', 'medium', 'large', 'xlarge' ];
var pge_breakpoint_widths = ['1-1', '1-2', '1-3', '2-3', '1-4', '3-4', '1-5', '2-5', '3-5', '4-5', '1-6', '5-6'];

//additional component code
fields: {
	pge_add_breakpoint_widths: {
		type: 'custom',
		name: 'Element Breakpoint Width',
		action: 'none',
		control: widthControl([{
			values: pge_breakpoint_widths,
		}]),
	},
//additional component code

var widthControl = function (settings) {

	//create the new control with the PgCustomPropertyControl constructor
	var width_control = new PgCustomPropertyControl('pge_width_control');

	width_control.onDefine = function () {
		//loop through each of the breakpoint_sizes
		for (var breakpoints_sizes = 0; breakpoints_sizes < pge_breakpoints.length; breakpoints_sizes++) {
			//generate a field name from each array value
			var field_name = 'pge_sizes_' + pge_breakpoints[breakpoints_sizes];
			//register each field with name and a set of controls
			this.registerInputField(field_name, createFieldDef(settings, pge_breakpoints[breakpoints_sizes]));
		};
	};

	width_control.onShow = function () {
		//create a table for our control
		var $table = $('<table/>');
		//add a blank cell for row label
		var html = '<td></td>';
		//add name labels for each column
		pge_breakpoints.forEach(function(size) {
			html += '<td><label>' + size + '</label></td>';
		});
		//add row with labels to the table
		var $row = $('<tr/>').html(html).appendTo($table);
		//add next row to table
		$row = $('<tr/>').appendTo($table);
		//add control lable
		var $td = $('<td/>').html('<lable>Breakpoint Widths</label>').appendTo($row);
		//loop through each of the break_point sizes
		for (var breakpoints_sizes = 0; breakpoints_sizes < pge_breakpoints.length; breakpoints_sizes++) {
			//create a data cell to hold each control field
			$td = $('<td/>').appendTo($row);
			//generate a field name from each array value - this should be identical to the registered name
			var field_name = 'pge_sizes_' + pge_breakpoints[breakpoints_sizes];
			//add each registered field to the created $td of the table
			this.showInputField($td, field_name, createFieldDef(settings, pge_breakpoints[breakpoints_sizes]));
		};
		//return the newly created control to the calling field
		return $table
	};
};

var createFieldDef = function (settings, width_name) {
	var span_select = {
		type: 'select',
		name: null,
		action: 'apply_class',
		show_empty: true,
```

```
                options: []
        };
        //Generate specific Key and Name values from array passed in the settings argument
        settings.values.forEach(function (value) {
                span_select.options.push({
                                                key: value + '@' + width_name,
                                                name: value
                });
        });
        //return the field control to the define method
        return span_select;
};
```

Component Field Structure

A field utilizing a custom control should assign the `type` key a value of `custom` and the `action` key a value of `none`. As mentioned above, the `control` key should receive a function that returns the new control. In order to keep components re-usable, it is often useful to pass an object containing multiple key:values to the custom control. For simplicity, in this example we are passing an object with a single key, `values`. That key receives an array `pge_breakpoint_widths`.

```
/additional component code
fields: {
        pge_add_breakpoint_widths: {
                type: 'custom',
                name: 'Element Breakpoint Width',
                action: 'none',
                control: widthControl([{
                        values: pge_breakpoint_widths,
                }]),
        },
//additional component code
```

PgCustomPropertyControl(control_id)

This constructor takes a single argument during instantiation, `control_id`. This `control_id` should be unique and best practice is to prefix with a plugin unique value to prevent conflict. It takes two key:value pairs, `onDefine` and `onShow`.

Example usage:

```
var widthControl = function (settings) {

        //create the new control with the PgCustomPropertyControl constructor
        var width_control = new PgCustomPropertyControl('pge_width_control');
        //remainder of widthControl function including .onDefine and .onShow keys
```

- onDefine

  This key acts as a hook that fires when the framework is first loaded. It receives a function that reserves the unique field names of each of the custom control elements using the constructor supplied `registerInputField`function. It typically is a loop that iterates over an array of values. Example usage:

```
var pge_breakpoints_array = [ 'small', 'medium', 'large', 'xlarge' ];

//additional function code
width_control.onDefine = function () {
                //loop through each of the breakpoint_sizes
                for (var breakpoints_sizes = 0; breakpoints_sizes < pge_breakpoints.length; breakpoints_sizes++) {
                        //generate a field name from each array value
                        var field_name = 'pge_sizes_' + pge_breakpoints[breakpoints_sizes];
                        //register each field with name and a set of controls
                        this.registerInputField(field_name, createFieldDef(settings, pge_breakpoints[breakpoints_sizes]));
                };
        };
//additional function code
```

In this example, we are creating a control that will allow the user to select the element width from a set of responsive widths (`pge_responsive_widths`)for four hypothetical breakpoints (`pge_breakpoints_array`). The anonymous function passed to `onDefine` will loop over an array named `pge_breakpoints_array`. For each of the items it will prefix with `pge_sizes_` and then pass that unique field name to the `registerInputField` function. In this example that will result in the reservation of 4 control slots, one for each

item in the `pge_breakpoints_array`.

- registerInputField(field_name, field_definition)

  This function takes two arguments:

  - **field_name** This argument receives a unique field name that is typically proceeded with a plugin-specific prefix.
  - **field_definition** This argument receives a function that returns an object containing the key:value pairs normally found in a component field. Conventionally, this object is returned by a function named `createFieldDef`, but can be any function that returns a unique field for each component in the custom control.

- on_show

  This key acts as a hook that fires when the component is selected in the tree and the custom control is shown in the properties tab. As a value it receives a function that returns HTML with each reserved control to display to the user. Each of the unique control ids reserved using the `registerInputField` function are passed to the `showInputField`.

```
width_control.onShow = function () {
                //create a table for our control
                var $table = $('<table/>');
                //add a blank cell for row label
                var html = '<td></td>';
                //add name labels for each column
                pge_breakpoints.forEach(function(size) {
                        html += '<td><label>' + size + '</label></td>';
                });
                //add row with labels to the table
                var $row = $('<tr/>').html(html).appendTo($table);
                //add next row to table
                $row = $('<tr/>').appendTo($table);
                //add control lable
                var $td = $('<td/>').html('<lable>Breakpoint Widths</label>').appendTo($row);
                //loop through each of the break_point sizes
                for (var breakpoints_sizes = 0; breakpoints_sizes < pge_breakpoints.length; breakpoints_sizes++) {
                        //create a data cell to hold each control field
                        $td = $('<td/>').appendTo($row);
                        //generate a field name from each array value - this should be identical to the registered name
                        var field_name = 'pge_sizes_' + pge_breakpoints[breakpoints_sizes];
                        //add each registered field to the created $td of the table
                        this.showInputField($td, field_name, createFieldDef(settings, pge_breakpoints[breakpoints_sizes]));
                };
                //return the newly created control to the calling field
                return $table
        };
```

- showInputField($el, field_name, field_definition)

  This function takes three arguments:

  - **$el** This argument passes a jQuery-wrapped DOM element indicating the location where the control should be inserted.
  - **field_name** This argument takes a unique field name, identical to the value passed to `registerInputField`.
  - **field_definition** This takes a function that returns the control field key:value pairs.

- createFieldDef()

  The function passed to `showInputField` conventionally receives this name, but any callback function name can be used. It returns a field array to the `onDefine` and `onShow` functions, including `type` and `action` key:value pairs.
  Example usage:

```
var createFieldDef = function (settings, width_name) {
        var span_select = {
                type: 'select',
                name: null,
                action: 'apply_class',
                show_empty: true,
                options: []
        };
        //Generate specific Key and Name values from array passed in the settings argument
        settings.values.forEach(function (value) {
                span_select.options.push({
                                                key: value + '@' + width_name,
                                                name: value
                });
        });
        //return the field control to the define method
        return span_select;
};
```

API Helper functions

Overview

Once your framework has been instantiated with descriptors there are a number of additional helpers and constructors made available through the API. Depending on the use-case only some of these may be needed. The most germane of these are listed below, with a short description of use case, arguments list, and example.

pinegrow.getCurrentProject()

This helper function returns an array of key:value pairs with information about the current project, including project folder URL. If the current page is not part of a project it returns 'null'. This helper is useful to test whether the current page is part of a project before trying to add resources.

pinegrow.getSelectedPage()

This helper function returns the CrsaPage variable for the current page, which contains an array of key:value pairs with information such as breakpoints and page file URL.

pinegrow.showQuickMessage(HTML, duration, single, error, don't_repeat_delay)

This helper allows the display of a message to the user. It is usually triggered by the `on_inserted` or `on_changed` keys. If the user has turned off quick notifications in the settings, the message will not be displayed unless `error` is set to true. If you need to display a message to the user whether or not quick message has been turned off, use `pinegrow.showAlert()` instead.

- HTML
  This argument receives an HTML or plain text string to display to the user
- duration
  This argument receives the number of ms to display the message. Note: this will block any interaction with the Pinegrow App for the duration of the message. Defaults to 2500ms.
- single
  This argument receives a boolean value and is optional. Generally only passed in if also passing a value to context. Has no impact on message display.
- error
  This argument receives a boolean value and is optional. If 'true' then the message modal will receive an orange border. To directly display an error message it is preferential to use `pinegrow.showQuickError()`.
- don't_repeat_delay
  This argument receives the number of ms to delay before allowing message to display again. For example, if you alert the user to refresh the page on element insertion and they are placing that same element on the page three times in succession, you would only want to display the message once, so the delay could be set to 60000ms (1 minute).

pinegrow.showQuickError(HTML, single)

This helper allows for the display of an error message to the user, which is a quick message box with orange background or border. It can be triggered from within a component or any time some check fails. For example, a particular element might only be used in the context of a project and give an error if added to a single page.

```
var isProject = pinegrow.getCurrentProject();
if (!isProject) {
        pinegrow.showQuickError('Save as a project first!');
} else {
  ...
}
```

- HTML
  This argument receives an HTML or plain text string to display to the user.
- single
  This argument receives a boolean value and is optional. Generally only passed in if also passing a value to context. Has no impact on message display.

pinegrow.showAlert(HTML, title, button_one_msg, button_two_msg, button_one_func, button_two_func)

This helper function receives up to 6 arguments (all technically optional) and opens a modal that collects user input. It can display up to two buttons, each with a different function.

- HTML
  This argument receives an HTML or plain text string to display as the body text of the modal. Optional and defaults to an empty string if null is passed.
- title
  This argument receives an HTML or plain text string to display as the title for the modal. Optional and defaults to 'Notice' if null is passed.
- button_one_msg
  This argument receives an HTML or plain text string to display as the first button text. Optional and defaults to 'OK' if null is passed.
- button_two_msg
  This argument receives an HTML or plain text string to display as the second button text. Optional with no button displayed if passed null.
- button_one_func
  This argument receives a function that is fired if the user clicks button number one. Optional and defaults to null if not passed. If the button is used to cancel any changes and close the modal and a second button with function is being passed, this argument should be passed 'null'.
- button_two_func
  This argument receives a function that is fired if the user clicks button two. Optional and defaults to null if not passed.

```
var $html = $('<h3>Do you wish to add the slider element script to your page?</h3>\
<p>This is required for the slider element to function.</p>');
pinegrow.showAlert ($html, 'Add slider JavaScript?', 'Cancel', 'Yes, please!', null, function(){
          addScript();
        });
```

pinegrow.getPlaceholderImage()

This helper will return a random image from the pinegrow server to serve as a placeholder in any image element. Note, this will only work if the user is connected to the internet.

Framework Hooks

on_project_loaded()

This key takes a function as value and calls the supplied function when a project with the plugin activated is loaded.

on_page_loaded

This key takes a function as value and calls the supplied function when a page with the plugin activated is loaded.

on_page_saved

This key takes a function as value and calls the supplied function when a page with the plugin activated is saved.

on_page_changed

This key takes a function as value and calls the supplied function when a page with the plugin activated is changed.

on_page_closed

This key takes a function as value and calls the supplied function when a page with the plugin activated is closed.

Next: Custom Controls (Custom%20Controls.md)

Menu Helpers

There are two major ways to add new menu items through the Pinegrow API. The first allows for addition of new menu items to the "Page" main menu using the framework key `on_page_menu` and passing in the menu items. The second allows for the addition of an entirely new menu using the `addPluginControlToTopbar()` helper function.

.on_page_menu(page, items)

This framework key is used for inserting additional items into the "Page" menu. The additional items will be added after the "Manage Google Fonts..." menu item and will only be visible when a page is opened in the editor. This key receives a function that is passed two arguments, `page` and `items`.

The `items` argument is an array of objects containing the items to be added to the menu. There are three different objects that `items` accepts.

1. An object with a single key:value pair
   `type: 'divider'`
   This object adds a horizontal divider in the dropdown menu.

2. An object that adds a header with two key value pairs:
   `type: 'header',`
   `label: 'Any string'`
   This object will cause Pinegrow to add a non-clickable header to the dropdown menu using the string passed as the `label` value.

3. An object with two key:value pairs:
   `label: 'Any string',`
   `action: function(){/*action to be performed when item is selected}`
   This object will display the `label` key value in the menu and execute the function passed in the `action` key value when the user clicks the menu item. Simple menu example

```
framework.on_page_menu = function (page, items) {
        items.push({
                type: 'divider'
        });

        items.push({
                type: 'header',
                label: 'New Menu'
        });

        items.push({
                label: 'Menu Item One',
                action: function() {
                        customFunction()
                }
        })
}
```

addPluginControlToTopbar(framework, $control, show_always, func)

This helper will create a new menu and insert it prior to the "Support" menu in the main header. It accepts four arguments, with the first two being required and the second two optional.

framework

The first argument is simply the framework object that we instantiated in our plugin. It defaults to "f" and is required.

$control

The second argument is the actual menu HTML and is required. It should be passed as a jQuery element. Each of the dropdown menus are individual `<li>` items. The menu is styled and enabled using Bootstrap 4 classes and toggles.

Simple menu example

```
var $control = $('
    <li id="example-menu" class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown"><span>New Menu Title</span></a>
        <ul class="dropdown-menu">
            <li><a href="#" id="item-one">Menu Item One</a></li>
            <li><a href="#" id="item-two">Menu Item Two</a></li>
        </ul>
    </li>
');
```

The next two keys work in tandem. One or both arguments must evaluate to true in order to display the new menu.

show_always

This argument takes a boolean value. If true, then the menu will be displayed whenever the framework is loaded, irrespective of page or project. Note, this is accomplished through appending the `$control` to the editor through jQuery manipulation. If this function is called to early in the load there will be an error due to the targeted on page element not existing.

func

This argument takes a function that evaluates whether the menu should be loaded and should return a boolean value. For example, it can be used to determine if there is a page currently open, or if it is part of a project.

Example of adding a menu only if there is a page open that is part of a project that utilizes our custom framework

```
pinegrow.addPluginControlToTopbar(framework, $menu, false, function () {
                var thePage = pinegrow.getSelectedPage();
                var theProject = pinegrow.getCurrentProject();

                if (thePage && thePage.hasFramework(framework) && theProject) return true;
                return false;
        });
```

Next: Helpers (Helpers.md)

Overview

Sections and fields are passed as component keys following the main body keys in the `PGComponentType` options. They add and organize controls for components and are optional if the component adds only an HTML snippet. Each section organizes a group of fields on either the Property or Actions tab of the Pinegrow Editor. Each field, in turn, encodes the control activity.

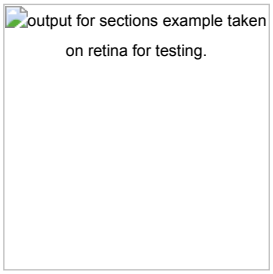
Sections Overview

The `sections` key receives an object of objects. Each object that it receives is a key:value pair with a unique name for key and an object for value. This object in turn has two required and one optional key:value pairs that define a set of controls.

Basic `sections` structure

```
//remainder of component code
sections: {
        unique_name_one: {
                name: 'Displayed Section Name One', \\required
                default_closed: true, \\optional
                fields:{...}, \\required
        },
        unique_name_two: {
                name: 'Displayed Section Name Two', \\required
                default_closed: false, \\optional
                fields: {...}, \\required
        }
}
//remainder of component code
```

Properties Panel from the above code when element is selected



Sections Key:Value pairs

---

name

This key takes a name that will be displayed in the properties or actions panel. Note that alongside this name the Pinegrow App will list the source of the control or action.

default_closed

This key receives a boolean. If set to true, the resulting section will be closed by default on first display. This key is optional.

fields

This key is again an object of objects. Each individual object is a control or action.

Fields Overview

Each field is a property control or action that modifies the element identified on the page by the main `selector` key. There are a number of built-in controls, such as checkboxes and drop-downs, but custom controls can also be built using API helpers. There are also several built-in actions, such as adding classes or attributes, but once again it is easy to build your own actions.

The `fields` key receives an object, that contains one or more individual field objects. Each individual field object has a unique id as key, with an object containing a number of key:value pairs as value.

Basic fields structure

```
//remainder of component code
fields: {
        field_one: {
                ...
        },
        field_two: {
                ...
        }
}
//remainder of component code
```

Fields Key:Value Pairs

---

name

This value for this key will be displayed in either the properties or actions tab next to the control, e.g. "Add dividers?" or "Display".

action

This key identifies what action Pinegrow should take when the user makes a selection with the control. It is used in conjunction with one or more additional keys

- apply_class
  This `action` value indicates that the user input or value of the `value` key should either be added or removed as a class on the element. This value can be supplied from a dropdown using the `options` key, from the textbox of a `text` type input, or from the `value` key when using a `checkbox` type.

- element_attribute
  This `action` value indicates that the value being supplied from the control should be either added or removed as an attribute of the element. This value can be supplied from the `attribute` key along with the `empty_attribute` key to produce an empty attribute, or a combination of the `attribute` key and `value` key or `select`, `text`, `image`, or `slider` user input.

- custom
  This `action` value indicates that a custom function, supplied by the `set_value` key, should be used to modify the selected element. Both `set_value` and `get_value` will be covered in the custom actions (Custom%20Actions.md) section.

An example of the `apply_class` action. This example would add the `btn-lg` class to the selected element when the user ticks the checkbox.

```
//remainder of component code
fields: {
        button_group_size {
                name: 'Make button large?',
                type: 'checkbox',
                action: 'apply_class',
                value: 'btn-lg'
        },
}
//remainder of component code
```

An example of the `element_attribute` action. This example will add the `data-shipping` attribute with a value supplied from a `select` dropdown.

```
//remainder of component code
fields: {
        shipping_method: {
                name: 'Shipping method?',
                type: 'select',
                options: [
                        {key: 'next-day', name: "Next Day"},
                        {key: 'two-day', name: "2 day"},
                        {key: 'standard', name: "Standard"}
                ],
                action: 'element_attribute',
                attribute: 'data-shipping',
                empty_attribute: false
        },
}
//remainder of component code
```

Element_attribute-specific key:value pairs

- **empty_attribute**
  This key takes a boolean value and determines whether or not the attribute being added to the DOM element requires a value. For example, in some cases an attribute of 'disabled' might be added to an element.

- **attribute_keep_if_empty**
  This key takes a boolean value. It is set to true if an attribute is valid whether or not it has a value. For example, the UIkit framework has an attribute `uk-sticky` that is valid as an empty attribute (essentially evaluating to true) or with data passed in like `uk-sticky="offset: 50"`.

---

type

This key takes a value that tells Pinegrow what type of control to display. The Pinegrow API has five main types built in:

Type	Output
checkbox	Displays a checkbox - basic boolean control
select	Displays a dropdown for selecting from a list of options - requires the options to be supplied either as an array of key:value objects, or as a function that returns the same
text	Displays a textbox to receive plain text/HTML/code
image	Displays a filepicker - can be used to select any file, not just images. For images it also displays a thumbnail
slider	displays a range slider for numerical input

In addition to the built-in values, the `type` key can also accept a value of `custom` to allow the control to be defined either using the `control` or `show` keys. This will be further covered in the custom controls (Custom%20Controls.md) section.

Checkbox-specific key:value:pairs

- **value**
  This key takes a value (numeric or string) that will be added or removed from the DOM element as class, attribute, or custom value when the checkbox is ticked.

- **negvalue**
  This key is optional and takes a value (numeric or string) that will be added or removed from the DOM element as a class, attribute, or custom value when the checkbox is unticked.

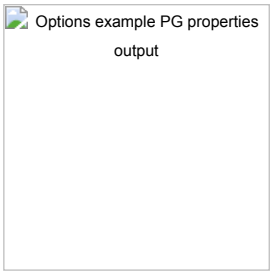
Basically, if a `negvalue` key is present, the checkbox will act similiar to the Jquery `.toggleClass()` and toggle between the two values. However, if the `negvalue` key value is already present on the element, it will **NOT** be removed on the first click. Instead, the `value` key value will be added, and on the next click it will behave as expected.

Select-specific key:value pairs

- **options**
  This key supplies the list of choices to be displayed in the `select` dropdown list. They are supplied as an array of choices, where each choice is an object with at least two key:value pairs.
    - name - this key takes a string that is displayed to the user in the select dropdown
    - key - this key takes a string that is the value returned when the user selects that option

```
//remainder of component code
fields: {
        pge_button_group_size: {
                name: 'Button Size',
                type: 'select',
                show_empty: true,
                options: [
                        {key: 'btn-group-lg', name: "Large"},
                        {key: 'btn-group-sm', name: "Small"},
                        {key: 'btn-group-xs', name: "Extra small"}
                ],
        },
}
//remainder of component code
```

Output of example above.



A third key `html` is used for making custom buttons in conjunction with the `toggle_buttons` key and will be covered in the custom controls (Custom%20Controls.md) section.

- **show_empty**
  This key takes a boolean value. If true, it will allow the user to select an empty value, or no value, rather than one from the list. Depending on action or other keys, this can have the effect of removing a particular class or attribute from the DOM element.

- **default_value**
  This key allows a default value from the options list to be pre-selected. As value it takes the value of the desired `key` from the options list. For example, in the above code you could add:

```
default_value: 'btn-group-lg',
```

Text-specific key:value pairs

- **live_update**
  This key takes a boolean value. If set to true, the targeted element will update in real-time as the user types. If set to false, the element will not update until the user hits enter.

- **placeholder**
  This key is semi-unique to the `text` and `slider` types. In the context of the `text` type it takes a string that will be displayed in the text box prior to user input. Note: this does not set a default value for the text box.

Image-specific key:value pairs

- **file_picker**
  This key takes a boolean value. If true it will display a folder icon at the right of the input box, along with a thumbnail if the selected file is an image.

Slider-specific key:value pairs

- **slider_min**
  This key takes a value for the lower end of the slider range.

- **slider_max**
  This key takes a value for the upper end of the slider range.

- **slider_step**

  This key takes a value for the amount each tick of the slider should increment the value.

- **slider_def_unit**

  This key is optional and takes a string that indicates the type of unit the slider represents, e.g. 'px', 'ms', 'deg'

- **placeholder**

  In the context of the `slider` type, this key takes a numeric value to display to the user prior to user input. Note, this does not set a default value for the field.

Simple slider example for setting section width from 20% to 100% in 5% increments.

```
//remainder of component code
fields: {
        pge_section_width: {
                name: 'Section width?',
                type: 'slider',
                slider_min: 20,
                slider_max: 100,
                slider_step: 5,
                slider_def_unit: '%'
        }
}
//remainder of component code
```

---

helptext

This key takes a plain text/HTML string that is displayed when the user hovers over the control display name.

on_changed

This key takes a function that receives 9 arguments in order - pgel, prop, value, oldValue, fieldDef, $field, eventType, fieldList, CrsaPage.

- **pgel**

  This argument is the pgParserNode (the source-code representation of the current DOM node) for the selected element.
- **prop**

  This argument returns the field name of the control that changed. In the previous example this would be 'shipping_method'.
- **value**

  This argument returns the value that was selected by the user.
- **oldValue**

  This argument returns the previous value that the user had selected. If there was no previous selection it returns 'null'.
- **fieldDef**

  This argument returns the full field definition, including `action` type and `name` key:value pairs.
- **$field**

  This argument returns a jQuery element representing the field control.
- **eventType**

  This argument returns the type of event that triggered the `on_changed` function - typically 'changed'.
- **fieldList**

  This argument returns an array of all the fields and field values for the section that the triggering field is included within.
- **CrsaPage**

  This argument returns an array of values that represent the selected page options. This includes the file URL, page name, attached stylesheets, and breakpoints.

An example of on_changed - this will add the selected class to the parent element:

```
//remainder of component
fields: {
          animation_toggle: {
                  type: 'select',
                  name: 'Add animation?',
                  action: 'apply_class',
                  show_empty: true,
                  options: [
                          { key: 'bounce', name: 'Bounce' },
                          { key: 'shake', name: 'Shake' },
                          { key: 'pulse', name: 'Pulse' }
                          ],
                  on_changed: function (pgel, prop, value, oldValue, fieldDef, $field, eventType, fieldList, CrsaPage){
                          var pgp = pgel.parent;
                          if (value) {
                                  pgp.addClass('animated');
                          } else {
                                  pgp.removeClass('animated');
                          }
                  }
          }
}
//remainder of component code
```

Conditional Field Display

show_if

This key determines whether a particular field will be displayed. It can receive as values either a standard if conditional that references another field in the same section, or a function that can reference multiple fields or other values from the element and returns TRUE if it should be shown. Functions receive two arguments in order, `values` and `pgel`.

- **values**
  This argument is an array of all the other fields in the section and their value.
- **pgel**
  This argument is the pgParserNode (the source-code representation of the current DOM node) for the selected element.
  Examples

```
def.sections = {
    wp_site_options : {
        name : 'Options',
        fields : {
            field_1 : {
                ...
                type: 'checkbox',
                value: 'ON'
            },
            field_2 : {
                ...
                show_if: "field_1" //or
                show_if: "field_1==ON" //if has specific value, or
                show_if: function(values) { return values['field_1'] == 'ON' && values['another_field']; }// field_1 has specific va
                show_if: function(values, pgel) { return values['field_1'] == 'ON' && pgel.hasAttr('data-name')}
            }
         }
    }
}
```

Next: Custom Actions (Custom%20Actions.md)

Adding Templates and Resources

Overview

Plugins can optionally offer a variety of resources to the user. This can include HTML, CSS, PHP, and JavaScript files, basically any file that you want to use in your project. These resources are copied from the plugin either upon activation of the library from the "Manage libraries & plugins..." menu or when a plugin HTML template is saved as a project.

Templates

Overview

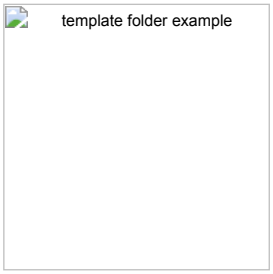
Templates are optional pre-made HTML pages that can include anything from a basic document, to a page that is already populated with a large amount of code. Typically, at minimum, a template will contain the basic HTML boilerplate, plus plugin specific CSS links (either local or remote) and/or JavaScript (either in page scripts or links to local or remote files). These templates are added using the `addTemplateProjectFromResourceFolder` constructor.

addTemplateProjectFromResourceFolder (template_folder, done, index, filter_func, skip_add_to_templates, absolute_folder)

This function is used when constructing a plugin that provides one or more templates as a base for the user. For example, the stock Bootstrap 4 framework provides 19 starter templates when the user clicks on "New page or project".

To utilize this in a plugin, create a folder that contains each HTML template file plus an additional subfolder that must be named "screenshots" and contain an image that is typically a screenshot of that template to display to the user. Each image must have the same name as the template file. Any other files that should be included when the template is saved as a project should also be in this folder. If desired, additional resources can be placed into a sub-folder(s).

Example template folder structure:



Arguments

- **template_folder** - the source of the template folder relative to the main plugin file - Note: if the folder is at the same level as the main plugin file you do not need to prefix with './' to search in the current directory.
- **done** - typically passed null, takes a callback function that is triggered once the framework is created and is passed the framework as an argument
- **index** - determines the order of framework display - typically set to 100 or higher to add the plugin framework at the end
- **filter_func** - receives a function that can edit the files added to the template - optional, typically used when adding multiple templates, each with specific resources
- **skip_add_to_template** - optional, internal development use
- **absolute_folder** - optional, internal development use

Simple usage:

```
framework.addTemplateProjectFromResourceFolder('template', null, 100);
```

Advanced usage

In this example there is a template specific CSS file ("advanced.css") that should only be added if a template ("advanced.html") requires that particular file. This same method can be used to include a specific JavaScript or other file.
First, add the additional template, CSS, and screenshot to the template folder.

Second, pass a function to the `addTemplateProjectFromResourceFolder` to select any files from an array of files (`templateSpecificFiles`) to exclude from the standard template. In this example, "advanced.css".

```
var templateSpecificFiles= ['advanced.css'];

framework.addTemplateProjectFromResourceFolder('template', null, 100, function (node) {
        var currentFilesName = templateSpecificFiles.filter(function (fileName) {
                        return node.name == fileName;
                });

                if (currentFilesName && currentFilesName.length > 0) {
                        node.required = false;
                };
})
```

This function is passed an argument, conventionally named node, that contains information about each added resource. This function returns any resources that match a filename included in the array of file names not to include automatically and then sets the `required` key of the resource to false. This ensures that the file will only be saved in the project if there is a link in the HTML template. This same function can also be extended to place the templates in a specific order using the `order` key.

```
var templatesOrder = ['index.html', 'advanced.html'];

        //Add our template into the project
        framework.addTemplateProjectFromResourceFolder('template', null, 100, function (node) {
                //remainder of function from above

                var templateIndex = templatesOrder.indexOf(node.name);
                if (templateIndex >= 0) node.order = templateIndex;
        });
```

Resources

Overview

Plugin resources are CSS and JavaScript files that can be loaded onto any page that activates the plugin. They can be provided as local files that get copied into the project folder, or as remote links that get added into the current HTML page(s). Additionally, resources can be used to upgrade existing project or page files and links.

PgComponentTypeResource (resource_url, code)

The PGComponentTypeResource constructor is used to add javascript and CSS files or links to a project. It recieves addition to the two arguments:

- **resource_url** This argument takes the URL of the file or folder to be added. For local files the URL can be generated using a helper function, `framework.getResourceUrl(path_to_resource)`.
- **code** This argument is typically not used and defaults to null.

getResourceUrl(path_to_resource)

This helper function takes the path to the resource file or folder relative to the base plugin file.

```
framework.getResourceUrl('template/resources/css/style.css');
```

In addition to the arguments passed in at instantiation it can accept a number of additional key:value pairs.

key	value
type	mime type - this allows the Pinegrow app to determine the correct way to embed the file on the HTML page, <script> and src for JavaScript, or <link> and href for CSS. Automatically set to correct mime type by lookup.
detect	regex string - determine if another file (like jQuery), or an earlier version exists on the page -- typically not used
footer	boolean - determines whether the item should be added to the head (false) or the bottom of the body (true) -- default: false
project	internal use only
isFolder	boolean - used to indicate if the resource is a folder or not -- default: false
source	url - typically the same as the resource_url -- default: null
relative_url	resource file location relative to the template -- default: null
replace	boolean (or function returning boolean) If the `detect` key is used and a match is found then this indicates if the found resource should be replaced with the file at resource_url -- default: false -- typically used to determine if a resource needs to be replaced during an update

The newly created resource object is returned as a value to the framework in the `resources` key.

```
framework.resources.add(new_resource)
```

Typical example code using file structure from above.

```
var resource_files = [
                'css/index.css',
                'js/my-js.js'
        ];
resource_files.forEach(function (resource_file) {
                var file = framework.getResourceUrl('template/resources/' + resource_file);
                var resource = new PgComponentTypeResource(file);
                resource.relative_url = resource_file;
                resource.source = file;
                resource.footer = resource_file.indexOf('.js') >= 0;
                resource.type = resource_file.indexOf('.js') >= 0 ? 'application/javascript' : 'text/css';
                framework.resources.add(resource);
        });
```

The `PgComponentTypeResource` constructor can also be used to add in remote files, such as CDN hosted framework files or Google fonts.

```
var resource = new PgComponentTypeResource('https://cdnjs.cloudflare.com/ajax/libs/d3/5.15.0/d3.min.js');
resource.type = 'application/javascript';
framework.resources.add(resource);

var resource = new PgComponentTypeResource('https://fonts.googleapis.com/css?family=Roboto&display=swap');
resource.type = 'text/css';
framework.resources.add(resource);
```

Framework key:value pairs for Templates and Resources

ignore_css_files

This key is used if the plugin includes customized CSS files within a template or as a stand-alone resource that shouldn't be altered by the user. It receives an array containing a single regex string (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) or multiple comma-separated regex strings. Any CSS file in the page resources that is matched by these string will be locked for editing under the stylesheet dropdown.

Single string

```
framework.ignore_css_files = [/my_plugin_style\.css/i];
```

Multiple regex strings

```
framework.ignore_css_files = [/my_plugin_style\.css/i, /my_other_styling\.css/i];
```

Next: Components (Components.md)

Creating Components

Overview

The `PgComponentType` constructor is the main way to add new snippets, property controls, and actions to the Pinegrow App. This constructor creates an object with a unique id, display name, and options that can be broken down into three sections.

1. A main body that contains key:value pairs that contain an optional HTML snippet, give information on how to identify the component for property controls or actions to target on the page, and data on how to display the element on the tree.
2. Objects passed through the `sections` key that identify one or more groups of property controls or actions.
3. Objects passed through the `fields` key that are individual property controls or actions.

How the component is added to the Pinegrow App depends on component type. For components that add property controls only, the component is added to the framework using `addComponentType` in conjunction with the `addPrefix(plugin_id)` function.

```
var my_property_control = new PgComponentType( 'my_control', 'My Control', {...} );
my_property_control.addPrefix(plugin_id);
framework.addComponentType(my_property_control);
```

Components that add reusable HTML snippets or actions require an additional constructor, `PgFrameworkLibSection`. This constructor creates a new panel that can be displayed in the Library or Actions Tab. Components for each panel are identified by passing an array of components using `setComponentTypes`. This object is then added to the framework object using either `addLibSection` to pass it to the Library tab, or `addActionsSection` to display it on the Actions Tab.

Library panel addition example

```
var pge_custom_lib_section = new PgFrameworkLibSection( 'pge_my_custom_section', 'My Custom Section');
pge_custom_lib_section.setComponentTypes([my_custom_component_one, my_custom_component_two]);
framework.addLibSection(pge_custom_lib_section);
```

or

Action panel addition example

```
var pge_custom_action_section = new PgFrameworkLibSection( 'pge_my_custom_section', 'My Custom Section');
pge_custom_action_section.setComponentTypes([my_custom_component_one, my_custom_component_two]);
framework.addActionsSection(pge_custom_action_section);
```

PgComponentType(unique_id, display_name,

This constructor is passed three arguments.

- **unique_id** An id that is unique within the plugin.
- **display_name** A name that is displayed in the library tab for the user to select to drag to the DOM. For components that add only property controls or actions this key can be passed an empty string.
- An object that contains the HTML, controls, and or actions. These options will be covered in more detail in the section of this document.

addPrefix(plugin_id)

This function is added as a key to the `PgComponentType` object. It adds prefixes to the unique_id of the component , as well as the sections and fields defined within the component. This ensures that there aren't any conflicts with other plugins. This must be called before addition of the component using `addComponentType`. It is passed a single argument.

- **plugin_id** An id unique to the plugin. It typically is the same as the plugin_id used when creating the framework.

```
var my_property_control = new PgComponentType( 'my_control', 'My Control', {...} );
my_property_control.addPrefix(plugin_id);
framework.addComponentType(my_property_control);
```

addComponentType(component)

This function adds the component defined by the `PgComponentType` constructor to the framework. It takes a single argument.

- **component** This argument is the name of the component object.

PgFrameworkLibSection(unique_id, display_name)

This constructor creates a new object for addition to either the HTML snippet library or the actions panels. This object is created after individual components are defined using the `PgComponentType` constructor. It receives two arguments.

- **unique_id** An id that is unique within the plugin and should be prefixed with the plugin_id to prevent conflicts.
- **display_name** A name that will be displayed as the title of the added section in the library or actions panel.

```
var pge_custom_lib_section = new PgFrameworkLibSection( 'pge_my_custom_section', 'My Custom Section');
```

setComponentTypes(components)

This function takes an array of component names (defined using the `PgComponentType` constructor) to be added to the section object created with the `PgFrameworkLibSection` constructor.

```
pge_custom_lib_section.setComponentTypes([my_custom_component_one, my_custom_component_two]);
```

addLibSection(section)

This function adds the passed section name to the library panel of the framework.

```
framework.addLibSection(pge_custom_lib_section);
```

addActionsSection(section)

This function adds the passed section name to the Actions panel of the framework.

```
framework.addActionsSection(pge_custom_action_section);
```

PgComponentType

Overview

The options object of the `PgComponentType` constructor can be further split into key:value pairs that provide the main body of the component, a `sections` object that organizes all of the controls or actions, and a set of `fields` objects within the sections object that contain the key:value pairs that describe each control or action. Each component has a single set of main body options, but can have multiple sections with multiple fields each.

Main Body Key:Value Pairs

selector

This key receives either a CSS selector(s) or function that uniquely identifies the element being created or controlled. This key is required and is passed 'null' if the component does not need to be identified by Pinegrow - for example, a sub-component of a larger HTML element. When receiving a CSS selector, that selector is evaluated by Pinegrow to return a boolean value. Any function passed through the `selector` key should return a boolean.

An example of a simple selector to target any page element with an attribute of 'pg-table'

```
selector: '[pg-table]',
```

An example targeting any element with a class of either 'container' or 'container-fluid'

```
selector: '.container,.container-fluid',
```

An example of a more complex selector using a javascript function. Note: the function can be passed a single argument that is conventionally named `pgel`. This argument contains the source-code representation of the current DOM node (pgParserNode). This example function checks to see if the tag of the node is NOT 'html', 'body', 'head', or 'script'. As a side note, `pgel.tagName` will always return lowercase, irrespective of the case in the document.

```
selector: function(pgel) {
        if (['html', 'body', 'head', 'script'].includes(pgel.tagName)) {
                return false;
        } else {
                return true;
        }
},
```

priority

Determines the order in which an action or property control component will display in the panel. The default is 1000. This key is optional.

code

This key receives any reusable snippet - HTML, Javascript, PHP. This key is optional. If you are putting together a component consisting of controls or actions only it doesn't need to be added.

An example passing both HTML and PHP.

```
code: '<h2>The title of this article is <?php the_title(); ?>.</h2>',
```

preview

This key receives a function that returns HTML representing what will be shown if the user hovers over the snippet in the library. This code is automatically generated from the `code` key:value pair. However, in the case of elements such as containers or rows it is useful to have a visual representation. This key is optional.

```
Other component code...
    preview: getGridPreview(),
remainder of component code...

var getGridPreview = function() {
        var white = 'height: 20px;';
        var blue = 'height: 20px; background-color: #D8E5F2;';
                return '<div class="container-fluid" style="border:2px solid #0098cc; height: 120px;">\
                        <div class="row">\
                                <div style="' + white + '"></div>\
                                <div style="' + blue + '"></div>\
                                <div style="' + white + '"></div>\
                        </div>\
                        <div class="row">\
                                <div style="' + white + '"></div>\
                                <div style="' + blue + '"></div>\
                                <div style="' + white + '"></div>\
                        </div>\
                </div>';
        }
```

This code results in the following being displayed when the user hovers over the element in the Library panel.

Note: If the `code` key has a value it will be appended to the preview HTML.

empty_placeholder

This key takes a boolean value. If true, it will insert a temporary placeholder element into the code until the user adds content. This is generally used for containers or rows to make them more easily dragged or targeted for editing.

```
empty_placeholder: true,
```

on_inserted

This key receives a function that is fired upon element insertion into the tree. This function takes two arguments, the DOM element (conventionally named 'pgel') and the page. One common use for this key is to display a message to the user or to refresh the page in the case of dynamic objects.

```
selector: '[slider]',
code: '...',
on_inserted: function(pgel, page) {
        pinegrow.showQuickMessage( 'Refresh page to see changes', 3000 );
},
...
```

on_changed

This key receives a function that is fired upon alteration of the element. This function takes two arguments, the DOM element and the page. It will also fire if one of the element's descendants changes. One common use for this key is to display a message to the user or refresh the page in the case of dynamic objects.

Next: Sections and Fields (Sections%20and%20Fields.md)

Creating a Plugin

Pinegrow plugins are Javascript functions that utilize the Pinegrow API to extend the capabilities of the editor. All Pinegrow plugins are instantiated by calling a function, either anonymous or named, once the Pinegrow App signals it is ready. This function gets passed the event and the `pinegrow` window variable as arguments.

```
$(function() {
    $('body').one('pinegrow-ready', function(e, pinegrow) {
        //plugin body
    });
});
```

or

```
$(function() {
    $('body').one('pinegrow-ready', function(e, pinegrow) {
        myFunctionName(pinegrow);
    });
});
function myFunctionName(pinegrow){
    //plugin body
};
```

Within the body of the plugin, a new framework object is created using the `PgFramework` constructor and passing in a unique plugin id that will be prefixed internally, along with a name for the framework.

```
var framework = new PgFramework(plugin_id, name);
```

This framework variable can then be populated with a variety of key:value pairs. Some of these pairs are informational, like `framework.author`, which will be displayed to the end user, or give parameters to the Pinegrow App about how the plugin is supposed to be managed or displayed. The most important of these are listed below in the section **framework descriptive elements**. Other pairs add the individual library components, items to the property panel, or actions panel. The most important of these are listed on the **Components** (Components.md) and **Sections and Fields** (Sections%20and%20Fields.md).

Typically, the descriptive key:value pairs for the framework are defined prior to returning the new object to the Pinegrow App using the `addFramework()` function.

```
pinegrow.addFramework(framework);
```

Framework Descriptive Elements

type

This key takes a value identifying the framework - usually the same as the plugin id passed into the framework, but it can also be used to delineate different versions or "types" of the same framework. For example, all of the included versions of the Bootstrap framework have a `type` of 'bootstrap', but a plugin id unique to their version - 'bs3.4.1' or 'bs4'. Defaults to the passed in key.

```
framework.type = 'key';
```

allow_single_type

This key takes a boolean value, usually "true", that prevents activation of multiple frameworks of the same `type`. Defaults to "false".

```
framework.allow_single_type = true;
```

description

This key takes an HTML or plain text string describing the plugin and can contain a link that is displayed when creating a new page using a template from the plugin.

```
framework.description = '<a href="http://my.website.com/">Custom plugin</a> that adds a really neat framework';
```

author

This key takes an HTML or plain text string with the author's name and is displayed when creating a new page using a template from the plugin.

```
framework.author = '<em>Pinegrow</em>';
```

author_link

This key takes a URL link that will be opened if the author name is clicked in the new page pop-up.

```
framework.author_description = 'https://pinegrow.com';
```

info_badge

This key takes a short string that will be displayed when creating a new page using a template from the plugin. This is useful for displaying version numbers, for example.

```
framework.info_badge = 'v1.0.0';
```

Typical example of basic framework instantiation:

```
$(function() {
    $('body').one('pinegrow-ready', function(e, pinegrow) {

        var plugin_id = 'pge';

        var framework = new PgFramework(plugin_id, 'Pinegrow Example');

                framework.type = plugin_id;
                framework.allow_single_type = true;

                framework.description = '<a href="http://my.website.com/">Custom plugin</a> that adds a really neat framework';
                framework.author = '<em>Pinegrow</em>';
                framework.author_description = 'https://pinegrow.com';
                framework.info_badge = 'v1.0';

        pinegrow.addFramework(framework);

    });
});
```

As shown below, this descriptor information will only be displayed when creating a plugin that adds a template for the user to select when starting a new page or project. It will not be displayed if the plugin only adds HTML snippets, actions, modifies workflow, or adds to the CMS.

Once your framework is created you can add optional templates and resources that are made available to the user. This is documented on the Templates and Resources (Templates%20and%20Resources.md) page. Additionally, you can now begin adding components (Components.md) and sections and fields (Sections%20and%20Fields.md) that control page elements.

Next: Templates and Resources (Templates%20and%20Resources.md)

Custom Actions

Overview

Custom actions do more to manipulate the DOM than adding a simple class or attribute. For example they can set multiple attributes and values at the same time, or add a class to a parent element in the DOM at the same time as adding one to the selected elements. They are comprised of two components, the `get_value` and `set_value` keys, with the `action` key set to `custom`.

get_value

This key accepts a function as a value. This function receives a single argument, `pgel`, which is the pgParserNode (the source-code representation of the current DOM node) for the selected element. This function should return the current value that the control should display. This value is based on the current state of the DOM. For example, if the custom control adds an attribute to the parent element, the `get_value` key function should fetch and examine the parent element and return "true" if the attribute is present. Depending on the type of control, the value that is returned will need to be different. |type|returned value| |---|---| checkbox | true/1 or null/false/0 select | value of `name` key text | user entered text image | file name slider | current slider value

set_value

This key accepts a function as value. This function receives two arguments, `pgel`, which is the pgParserNode (the source-code representation of the current DOM node) for the selected element, and `value`, which is the user selected value from the field. It is the responsability of this function to manipulate the DOM in response to the user selection.

Simple example of a custom field

```
//other component code
fields: {
        pge_add_tooltip: {
                type: 'checkbox',
                name: 'Add tooltip?',
                action: 'custom',
                value: '1',
                get_value: function (pgel) {
                        return pgel.getAttribute('data-toggle') === 'tooltip' ? '1' : null;
                },
                set_value: function (pgel, value) {
                        if (value) {
                                pgel.setAttribute('data-toggle','tooltip');
                                pgel.setAttribute('data-placement', 'top');
                        } else {
                                pgel.removeAttribute('data-toggle');
                                pgel.removeAttribute('data-placement');
                        }
                        return value;
                }
        }
}
```

In an actual plugin an additional plugin to select the value of the `data-placement` attribute would be conditionally displayed when a tooltip was added. Note: along with the `custom` action, a `value` key with a value that is non-zero and defined is required. In this example the `value` key is passed a string, `'1'` but passing a non-zero integer, or any string is sufficient.

Next: Menus (Menus.md)